



Advanced Control Systems Lab

ACSL TurtleBot3 e-Manual

Adwait Verulkar

Yi Han

Walter Peregrim

Casper Gleich

adwaitverulkar@vt.edu

hanyi@vt.edu

peregrim@vt.edu

casper1@vt.edu

June 2020

Contents

Abstract	2
1 Overview and Setup	3
1.1 Remote PC Setup	3
1.1.1 Quick Installation using Shell Script (recommended)	3
1.1.2 Manual Installation of ROS Melodic	3
1.1.3 Network Configuration	4
1.1.4 Compiling Source Code on Remote PC	4
1.2 Raspberry Pi SBC Setup	4
1.3 OpenCR Setup	5
2 Setting up Wireless File Sharing on an SBC Running Ubuntu	7
3 Replacing the Raspberry Pi with an alternate SBC	8
3.1 Hardware Changes	8
3.2 Software	8
3.3 ROS	9
4 Syncing Time using NTP Server and Client	10
4.1 Configure NTP Server on the Remote PC	10
4.2 Configure NTP Client on the SBC	11
5 Bring up the Turtlebot	12
6 Repository Introduction	14
7 Torque Control using Dynamixel	15
8 Turtlebot Graphical User Interface	27
8.1 Qt	27
8.2 GUI Widget Walkthrough	28
8.2.1 Model Configuration	29
8.2.2 Controller Configuration	30
8.2.3 2d Navigation Goal	30
9 List of ROS Nodes, Topics, Parameters, Transform	32
9.1 List of Important ROS Nodes	32
9.2 List of Topics and Their Types	32
9.3 List of Important Topics and Descriptions	33
9.4 List of Important Parameters and How to Find Them	33
10 Transferring PID Controller to SBC [Experimental Future Work]	38

Abstract

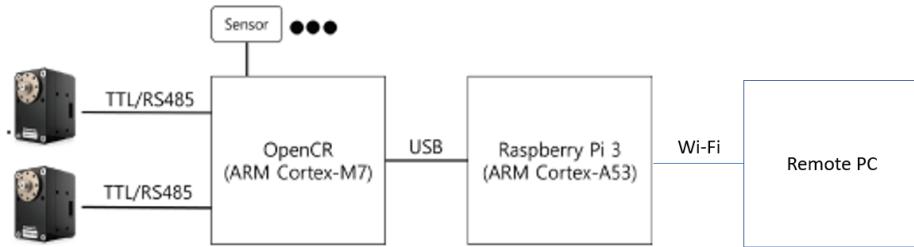
This guide contains the documentation for the customized **TurtleBot** developed for Advanced Control Systems Laboratory. The purpose of this Summer 2020 project was to deliver a modular reconfigurable PID controller for the Turtlebot wheel actuators to further the research of ACSL on SLAM, Navigation and Control Algorithms. The entire software has been written using ROS (`roscpp` and `rospy` libraries) and is present on 3 different machines. The software is based on the original code written by ROBOTIS for a stock Turtlebot3. The **Turtlebot3 eManuel** for ROS1 is the best starting point to get an initial understanding of the software.

This guide is not meant to be a comprehensive tutorial for ROS, Ubuntu Shell, Git or any tools used for development like Arduino IDE, Visual Studio Code etc. Understanding of these tools is assumed to be a pre-requisite, and instructions on how to "Bring Up" a new **TurtleBot3** have been provided. There exists a detailed description of the customizations done to the stock code, without repetitions from external literature. For content outside the guide, links have been provided at the appropriate places for further reading. However, this guide should be enough to get a new **TurtleBot3** up and running with ACSL Software.

1 Overview and Setup

TurtleBot3 is a small ROS-based mobile robot. The TurtleBot3 can be customized by changing the source code and addition of new hardware. The TurtleBot3 model used in this project is Waffle Pi. It is equipped with a Raspberry Pi Camera Module (v2), a 360-degree Laser Distance Sensor (LDS) and Dynamixel XM430 210-T motors.

TurtleBot3 architecture is described in the following image.



The four major components of TurtleBot3 are

1. Raspberry Pi 3B+ Single Board Computer (SBC).
2. OpenCR Embedded System Board.
3. LDS and Camera Sensors.
4. Dynamixel Actuators.

Raspberry Pi has a WiFi module built onto the board which connects to a computer, known as the Remote PC. Remote PC is used to send goal waypoints for SLAM and Navigation to TB3. Visualization of the robot is also performed on the remote PC using a ROS-based tool called `rviz`.

There is a one time setup that needs to be performed on the Remote PC, Raspberry Pi and OpenCR board. The following sections detail the procedure to setup the software on various components.

1.1 Remote PC Setup

To begin you will need to install the appropriate ROS version on your remote PC. The remote PC will be running ROS Melodic (EOL date: May 2023).

1.1.1 Quick Installation using Shell Script (recommended)

A shell script `install_melodic.sh` has been created to ease the installation of ROS and dependent packages on the Remote PC. This is located on the GitHub repo and can be cloned from there. Open a terminal and run the following commands.

```
git clone https://github.com/hanyiabc/ACSL_turtlebot3.git
cd ACSL_turtlebot3/
chmod 755 ./install_melodic.sh && bash ./install_melodic.sh
```

If you face any errors that cannot be resolved, the `install_melodic.sh` file is readable and you can execute those commands individually.

1.1.2 Manual Installation of ROS Melodic

If the shell script installation doesn't succeed, you can also follow instructions on ROS Wiki for setting up ROS Melodic on Ubuntu 18.04 manually. Complete the instructions on the following website.

<http://wiki.ros.org/melodic/Installation/Ubuntu>

Then, install required dependencies by running the last 3 commands in the `install_melodic.sh` file. They can be easily found under the "Install Dependencies" section.

Finally clone the repository using the following commands.

```
git clone https://github.com/hanyiabc/ACSL_turtlebot3.git
```

1.1.3 Network Configuration

Steps for configuring the wifi on the remote pc are taken from the following site and requirement specific instructions have been mentioned below.

```
https://emanual.robotis.com/docs/en/platform/turtlebot3/pc\_setup/#network-configuration
```

For this project we will use the Linksys Router on the **Linksys04294** network. It is helpful to write down the IP address of the remote pc, since you will need it later to setup the turtlebot pc network settings.

1.1.4 Compiling Source Code on Remote PC

Once the network is setup on the Remote PC, go ahead and build the GitHub repository. This can be done by navigating to the top level folder of the repository **ASCL_turtlebot3** and running `catkin_make` in the terminal.

Finally add the environment variable for the **TurtleBot3** model to `.bashrc` file and source it. Possible values are *waffle*, *waffle_pi*, *burger*. We have our own customized model and hence we have defined our own model name. Run the following commands to add the variables to the `.bashrc` file.

```
echo "export TURTLEBOT3_MODEL=waffle_pi_effort_controller" >> ~/.bashrc
source ~/.bashrc
```

A different URDF file (an XML file for the ROS robot model) is loaded based on this environment variable. For this project however, we have defined new URDF file for effort control. By adding the above environment variable, ROS would use the appropriate file to model the robot for visualization and take the appropriate geometrical dimensions/parameters in the code.

This completes the Remote PC setup. It is advised to open `.bashrc` file using **Gedit** or any other text editor to make sure there are no duplicate commands being run. Having duplicates doesn't lead to any errors, but some variables might get redefined and it is not considered good practice to have multiple commands for setting up the same environment variables or aliases.

1.2 Rasberry Pi SBC Setup

The single board computer on **TurtleBot3** will be running ROS Kinetic (EOL date: April 2021) for the ease of setup since ROS Melodic requires manual compilation when using Raspbian. No problem has yet been found by us on the communication between these 2 releases. Now it is time to look at the turtlebot SBC. For the turtlebot 3 Waffle Pi model, the on board PC is a raspberry PI. This step requires a micro SD card with adapter, a monitor with an HDMI input, a USB keyboard and mouse, and a power source for the turtlebot. ROBOTIS provides a prebuilt desktop environment for the turtlebot with ROS kinetic. Instructions for flashing this distribution can be found here. (**Step 6.2.1.2 Install Linux Based on Raspbian, Do not use the other two methods.**).

```
https://emanual.robotis.com/docs/en/platform/turtlebot3/raspberry\_pi\_3\_setup/#raspberry-pi-3-setup
```

To use the Raspberry PI without a physical access, use SSH. Example SSH command:

```
ssh pi@YOUR_RASPBERRY_PI_IP
```

For the ease of use, VNC configuration is recommended. This provide remote desktop functionality. In case of Debian Stretch (the version that the manual will use), install Real VNC by running this command on the Raspberry PI:

```
sudo apt update
sudo apt install realvnc-vnc-server realvnc-vnc-viewer
```

Then enable VNC server by

```
sudo raspi-config
```

Then navigate to Interfacing Options. Scroll down and select VNC - Yes.

Ubuntu Desktop 18.04 comes with VNC client Remmina. On Windows, use VNC viewer <https://www.realvnc.com/en/connect/download/viewer/>

1.3 OpenCR Setup

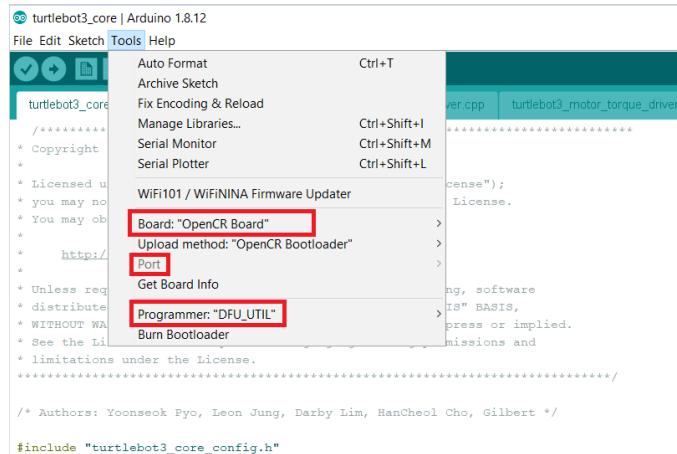
Most of the software is located in the form of a binary file that is burnt to the EEPROM of the STM32F746 chip on the OpenCR board. As the software used for ACSL projects is custom, it is burnt using Arduino IDE.

The original source code for TurtleBot3 used the inbuilt PID controller of the Dynamixel XM430 210-T actuators. However, to benchmark the performance of the controllers, the control is shifted from the motors to higher level software, which is a custom ROS PID node in this case.

Arduino IDE can be used to burn the `turtlebot3_core.ino` file located in `ACSL_turtlebot3/src/turtlebot3_core` directory. Instructions for setting up the Arduino IDE for TurtleBot3 can be found in the following link.

<https://emanual.robotis.com/docs/en/parts/controller/opencr10/#arduino-ide>

Once Arduino is setup, launch it and check if the port settings are correctly configured.



Connect OpenCR via microUSB to the PC and flash the firmware using this button.



2 Setting up Wireless File Sharing on an SBC Running Ubuntu

This section is optional. If there are frequent changes to the OpenCR firmware then it makes more sense to flash it using the SBC. However, as there is no internet access in the lab for the TurtleBot SBC, the required binary files need to be transferred over WiFi. The software we will be using to share files via WiFi is called Samba. To install it follow these steps on the selected SBC.

```
sudo apt update  
sudo apt install samba
```

Next, pick the directory you would like to share and copy its path. In this example I will use the **/odroid/home** directory. Now open the samba configuration folder.

```
sudo nano /etc/samba/smb.conf
```

Add the following to the end of the document. **"/odroid/home"** should be entered with the desired share directory.

```
[sambashare]  
comment = Samba on Ubuntu  
path = /odroid/home  
read only = no  
browsable = yes
```

Press Ctrl-X, Y, and Enter to save.

Next add the firewall rules for Samba.

```
sudo service smbd restart  
sudo ufw allow samba
```

Finally setup a user for the Sambashare. This will need to match the name of an account already on the SBC. For this example I will use the **“Odroid”** user with password **1234**.

```
sudo smbpasswd -a Odroid  
Please Enter Password: 1234  
Retype Password : 1234
```

The folder should now be accessible from any device on the same network. To access it go to **file manager, other locations** and enter this network address

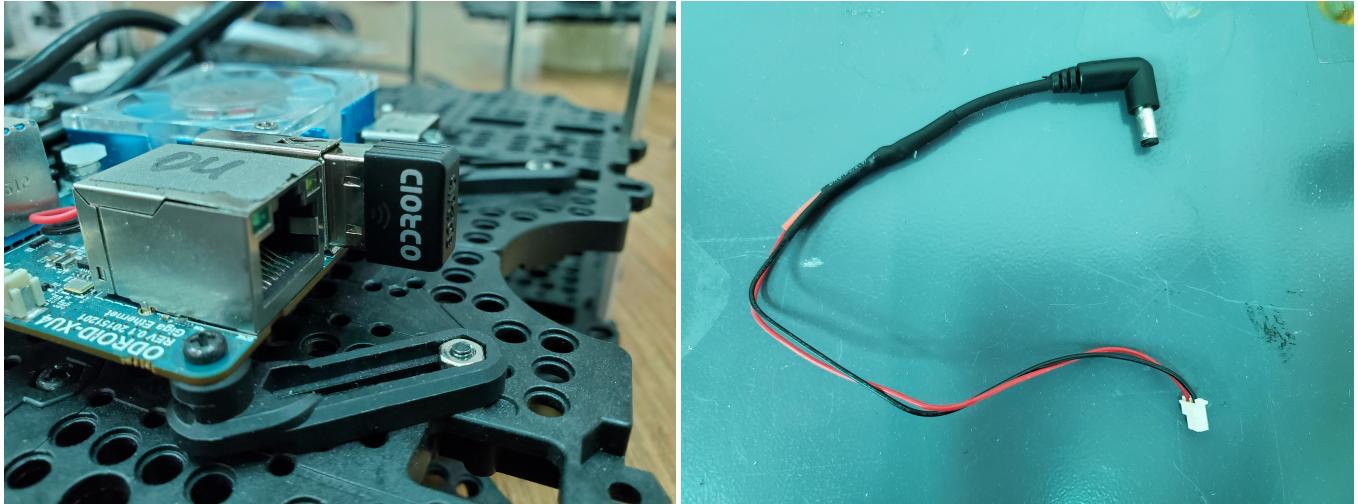
smb://(ip-address of sbc)/sambashare

A prompt will appear for entering the user name and password you just setup.

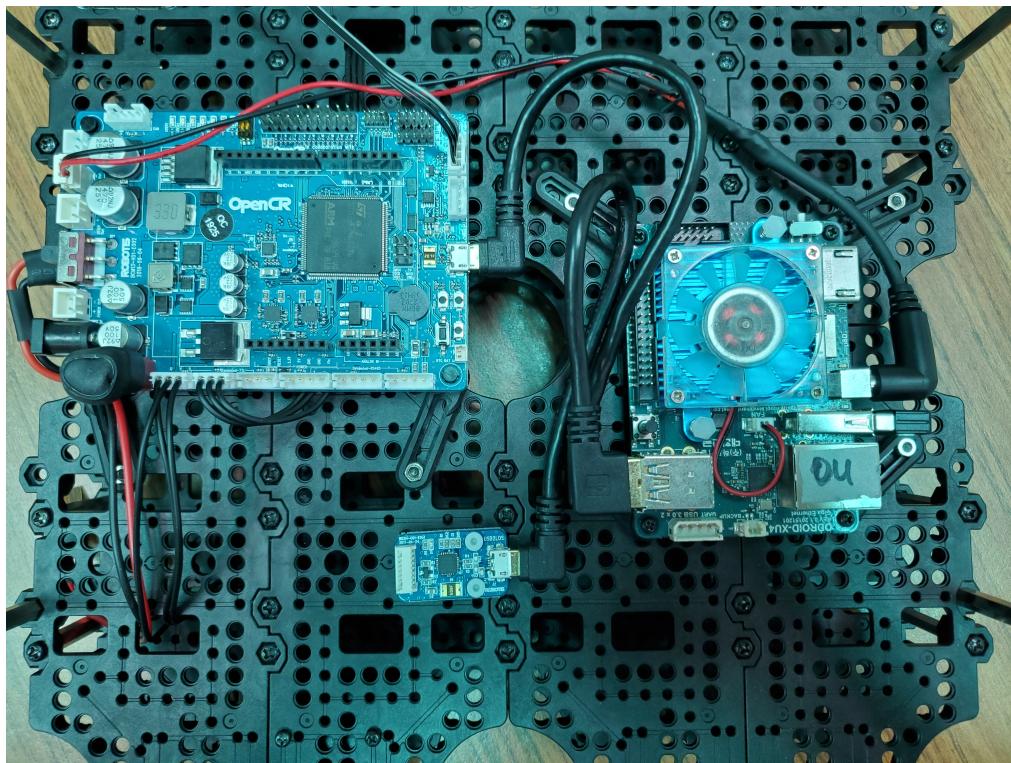
3 Replacing the Raspberry Pi with an alternate SBC

3.1 Hardware Changes

Before starting, you will need to attach an additional USB wi-fi device, as well as modify the OpenCR power output cable to fit the J5 barrel connector used by the Odroid. The 5v output port used for the Raspberry Pi will work for the Odroid if the connector is modified.



The Odroid XU4 has 2 modes for booting from memory. The micro SD card and the eMMC memory chips. There is a switch near the power port for switching between these 2 modes



3.2 Software

First you will need to flash a bootable ubuntu 16.04 OS onto the chosen memory device.
A suitable version can be found at

https://odroid.in/ubuntu_16.04lts/

We used the Ubuntu Mate version **ubuntu-16.04.3-4.9-mate-odroid-xu4-20171025.img.xz**

Connect the chosen memory device, and open a flash program such as Etcher and wait for flashing to finish.

After reconnecting the memory device, you can power up the Odroid. .The default user name is **Odroid**, and default password is also **Odroid**.

3.3 ROS

Follow the ROS wiki instructions for installing Ros-Kinetic.

<http://wiki.ros.org/kinetic/Installation/Ubuntu>

Next install the dependent packages using steps (3) through (5) of 6.2.1.1 Install Linux (Ubuntu MATE) from the turtlebot E-manual.

https://emanual.robotis.com/docs/en/platform/turtlebot3/raspberry_pi_3_setup/#raspberry-pi-3-setup

After finishing this step, you will be able to connect to the Odroid via SSH on the remote PC using the command line ssh odroid@192.168.xxx.xxx and using password odroid.

4 Syncing Time using NTP Server and Client

Raspberry Pi and Odroid SBCs do not have a hardware clock, and therefore it relies on an active internet connection to fetch the accurate time from Ubuntu time server. However, this is not possible as Linksys router, does not have internet access.

Every time SBC boots up, the time has to be updated. The time on the Remote PC and the SBC have to be synced up in order for the various transformations to work for the purpose of running SLAM and Navigation in the ACSL Lab Environment. Hence, to sync up time between these two machines, NTP protocol is being used.

NTP or Network Time Protocol is a protocol that is used to synchronize all system clocks in a network to use the same time. When we use the term NTP, we are referring to the protocol itself and also the client and server programs running on the networked computers. NTP belongs to the traditional TCP/IP protocol suite and can easily be classified as one of its oldest parts.

When you are initially setting up the clock, it takes six exchanges within 5 to 10 minutes before the clock is set up. Once the clocks in a network are synchronized, the client(s) update their clocks with the server once every 10 minutes. This is usually done through a single exchange of message(transaction). These transactions use port number 123 of your system.

In this section, a step-by-step procedure is described on how to:

- Install and configure the NTP server on a Ubuntu machine.
- Configure the NTP Client to be time synced with the server.

These instructions are taken from <https://vitux.com/how-to-install-ntp-server-and-client-on-ubuntu/>. However, it is advised that the following instructions are followed as there are some project specific changes that have been made.

4.1 Configure NTP Server on the Remote PC

Open Terminal and run the following commands **on the remote PC**

```
sudo apt-get update  
sudo apt-get install ntp
```

Most likely NTP will already be installed on the machine. After installation, the configuration file needs to be altered and server pools closest to the PC location needs to be added. This step is optional, as most likely the servers will be configured correctly.

```
sudo gedit /etc/ntp.conf
```

Replace/edit the server pools with the following.

```
pool 0.us.pool.ntp.org iburst  
pool 1.us.pool.ntp.org iburst  
pool 2.us.pool.ntp.org iburst  
pool 3.us.pool.ntp.org iburst
```

Save and close the configuration file. In order for the above changes to take effect, you need to restart the NTP server. The time from these servers will be synced with the time on the remote PC. as it has internet access. This will make sure that the NTP server clock is accurate.

```
sudo service ntp restart
```

Run the following command to check if the NTP server is configured properly. You should see the remote PC connecting to various servers.

```
sudo service ntp status
```

Finally, the system's UFW firewall needs to be configured so that incoming connections can access the NTP server at UDP Port number 123. This is the most important step, as it will configure the remote PC to allow client connections.

```
sudo ufw allow from any to any port 123 proto udp
```

Now the Ubuntu host machine is configured to be used as an NTP server. Please take a note of the IP address of the remote PC when it is connected to the appropriate router. IP address of a machine depends on which router it is connected to, and hence is not constant. We will need this address in the next step.

4.2 Configure NTP Client on the SBC

The `ntpdate` command will let you manually check your connection configuration with the NTP-server. Only in this case, the server would be the one that we specify. It should already be installed on the SBC, but in case it is not, run the following command **on the SBC**.

```
sudo apt-get install ntpdate
```

As the SBC will be a client to the host we set up on the remote PC, we need to modify the hosts file for NTP.

```
sudo gedit /etc/hosts
```

Now add your NTP server's IP (remote PC IP address noted in the previous step) and specify a hostname. Assuming that the remote PC IP address is 192.168.1.183 and the server has to be named **ACSLServer**, add the following line to this file:

```
192.168.1.183      ACSLServer
```

Save and close the NTP hosts file. Because we want our client to sync time with the NTP server, the `timesyncd` service on the client machine needs to be disabled by running the following command.

```
sudo timedatectl set-ntp off
```

Now, we need to specify this NTP server on the remote PC in the NTP configuration file of the client. If NTP is not installed install it using the following command.

```
sudo apt-get install ntp
```

Now we want our client machine to use our own NTP host server to be used as the default time server. For this, we need to edit the `/etc/ntp.conf` file on the client machine.

```
sudo nano /etc/ntp.conf
```

Add the following line to the file before the list of other servers. There should be a comment indicating the location where new servers can be added, although this is only for readability and functionality shouldn't be affected.

```
server ACSLServer prefer iburst
```

Finally restart the NTP service and the time should be synced.

```
sudo service ntp restart
```

Now your client and server machines are configured to be time-synced. You can view the time synchronization queue by running the following command:

```
ntpq -p
```

If **ACSLServer** is seen in the list of servers then the client is properly configured.

Finally the timezone needs to be set on the SBC to **America/New-York**. On the Odroid XU4 running Ubuntu with MATE desktop, it is a simple command line instruction as follows.

```
sudo timedatectl set-timezone America/New-York
```

This command may not work on the Raspberry Pi 3B+, but the timezone can be changed using the `raspi-config` utility.

Once the timezones are set, the SBCs are configured properly.

5 Bring up the Turtlebot

Once the remote PC and Raspberry PI are setup according to the instructions provided in the previous sections, Follow the instructions below to bring up turtlebot.

On Remote PC

```
roscore
```

On SBC

```
roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

On Remote PC, once you followed the instructions for building the workspace and setting up the environment variables, load the environment variables related to this workspace (Note: This has to be done everytime you want to run something from this workspace from a new terminal)

```
cd ASCL_turtlebot3/
source devel/setup.bash
```

To run teleop to control the Turtlebot with keyboard.

```
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

To run the navigation with PID velocity effort controller and SLAM (with localizatin provided by SLAM and no initial map required) on the physical robot.

```
roslaunch turtlebot3_bringup turtlebot3_physical_nav_control.launch
```

Note that the user might need to re-sync the time using NTP between the SBC and remote PC. Since SBC is not powered during shutdown, the time might be out of sync and this will cause error on the TF package from ROS since ROS uses time to track the transformations between each frames. Thus causing problem to the navigation stack. Simply do a manula update using NTP before lanuching will solve this issue.

To manually update the time and date using NTP (run this on the SBC instead of the remote PC and replace IP_ADDRESS_OF_THE_REMOTE_PC with the IP address of the remote PC):

```
ntpdate -u IP_ADDRESS_OF_THE_REMOTE_PC
```

To run the navigation with PID velocity effort controller and SLAM (with localizatin provided by SLAM and no initial map required) on Gazebo for simulation.

```
roslaunch turtlebot3_bringup turtlebot3_sim_nav_control.launch
```

To run SLAM only to draw maps Choose a pre-defined Gazebo world from the following:

- turtlebot3_empty_world.launch
- turtlebot3_world.launch
- turtlebot3_house.launch
- turtlebot3_stage_1.launch
- turtlebot3_stage_2.launch
- turtlebot3_stage_3.launch
- turtlebot3_stage_4.launch

For example, if **turtlebot3_world.launch** is chosen, run the following command to bring up a virtual Turtlebot on Gazebo for simulation

```
roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

Then, run the following commands to launch the SLAM nodes and remote control using keyboard.

```
roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping  
roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

Once you are satisfied with the map, save the map by running

```
rosrun map_server map_saver -f ~/map
```

6 Repository Introduction

The repository is a catkin workspace. A catkin workspace has a src folder that contains multiple packages. The worth-mentioning ones are:

- hls_lfcd_lds_driver
- raspicam_node
- ros_control
- simulink
- turtlebot3
 - turtlebot3_bringup
 - turtlebot3_navigation
 - turtlebot3_slam
 - turtlebot3_teleop
- turtlebot3_core
- turtlebot3_setup_motor
- turtlebot3_simulations
- turtlebot3_gazebo

Not all packages are catkin package. **simulink**, **turtlebot3_core** **turtlebot3_setup_motor** are all just directories. All the rests are properly configured as catkin packages, meaning all the nodes defined under the package will be compiled when running the **catkin_make** command.

hls_lfcd_lds_driver and **raspicam_node** are drivers for the PI Camera and lidar. These will be running in the single board computer.

ros_control contains all the configuration and nodes necessary to control the robot. It contains the PID controller configurationl, PID gains, etc. Two launch files are included for controlling the robot in either the simulations or in the physical world. They are:

- turtlebot3_control.launch
- turtlebot3_control_simulation.launch

These launch file are not ran directly, instead they are incluced by other launch file that we will introduce later. The PID gains can be tuned by either changing the parameters in the launch file or use a ros package called rqt_reconfigure to adjust parameters with sliders at runtime. There are 3 nodes under the **ros_control** package. They are **differential_driver.py**, **ros_control_odom_pub** and **ros_control_node**. The **differential_driver.py** is a Python ros node that split the incoming /cmd_vel messages into left and right velocity for the differential drive robot based on vehicle geometry. The **cmd_vel** is the standard topic that ROS nav stack used to output the velocity command. It commands the robot in terms of translational velocity and orientation. The **ros_control_node** is a C++ ros node compiled from the soruce file: **message_redirect.cpp**. This node simply takes the left and right wheel angular velocities feedbacks from **joint_states** converts them to linear velocities and publish them into 2 Float64 messages for the PID controller to read. The **ros_control_odom_pub** is a C++ ros node compiled from the source file: **odom_publisher.cpp**.

The directory **simulink** contains all the MATLAB scrips and Simulink models for controlling the robot with MATLAB (currently not uisng MATLAB).

turtlebot3_bringup contains all the launch file for bringing up the Turtlebot in the SBC and the remote PC. **turtlebot3_physical_nav_control.launch** was added to the bringup package to provide a convinient one-file launch that will handles everything. This launch file brings up the turtlebot on the remote PC, runs the velocity control and the navigation stack with SLAM. A simulation version will be provided in the future.

turtlebot3_description contains the URDF and Gazebo descripcion of the turtlebot for both simulation and physical robot.

For Waffle Pi, **turtlebot3_waffle_pi.gazebo.xacro** and **turtlebot3_waffle_pi.urdf.xacro** are provided by the turtlebot package, however, for effort control, we added 2 more files. They are:
turtlebot3_waffle_pi_effort_controller.gazebo.xacro and **turtlebot3_waffle_pi_effort_controller.urdf.xacro**
These files are derived from the original descriptions and added ability to control effort in both the simulation and the physical robot.

turtlebot3_navigation contains the default configuration for running the ROS navigation stack with the Turtlebot. A new launch file **turtlebot3_navigation_no_map.launch** was created. This launch file derived from the original navigation launch file, but the difference is that this configuration doesn't need a SLAM map for localizatin. The localizatin is provide by SLAM instaed of Adaptive Monte Carlo Localizatin. The default launch file **turtlebot3_navigation.launch** will run navigation assuming that a SLAM map has been drawn by running SLAM before. In this configuration, localizatin is provided by AMCL.

turtlebot3_slam contains all the configurations and launch files related to running SLAM. Several SLAM algorithms are pre-configured and supported but only Gmapping and Cartographer are tested, feel free to test other SLAM algorithms.

turtlebot3_teleop containes the node and launch file necessary for keyboard teleop (using keyboad to control the turtlebot).

turtlebot3_core derives from the original package from the OpenCR library. This is an Arduino sketch and this program is running in the OpenCR microcontroller. This package handles every low-level operation of the physiscal turtlebot. That includes reading sensor information, commanding motor torque, reading encoder information, etc.

turtlebot3_setup_motor is also an Arduino sketch that needs to run on the microcontroller, however, this only need to be ran once for each new set of Turtlebot motors. This package derives from the original **turtlebot3_setup_motor** from the OpenCR library. The original library sets the servos to velocity control operating mode. This package added the options to set the servos to torque control opearting mode.

turtlebot3_gazebo in located under **turtlebot3_simulations**. This package contains all the Gazebo related files necessary for running simulation with Gazebo. It contains a few pre-configured Gazebo world and 3D model for each model of the turtlebot.

7 Torque Control using Dynamixel

Dynamixel is a microcontroller based actuator with in-built PID controller. OpenCR communicates with Dynamixel using packet communication via TTL/RS45 ports.

Hardware level abstraction is achieved via Dynamixel SDK library available in several programming languages. (CPP used in our case.) Dynamixel register addresses and byte sizes for both RAM and EEPROM memory provided in control table. RAM is most frequently used memory for robot applications, while startup settings are stored on EEPROM.

The Control Table is a structure that consists of multiple Data fields to store status or to control the device. Users can check current status of the device by reading a specific Data from the Control Table with Read Instruction Packets. WRITE Instruction Packets enable users to control the device by changing specific Data in the Control Table. Packet sizes range from 1 – 4 bytes.

Following is a snapshot of the Dynamixel EEPROM Control Table. Each data in the Control Table is restored to initial values when the device is turned on. Default values in the EEPROM area (addresses 0-63) are initial values of the device (factory default settings). If any values in the EEPROM area are modified by a user, modified values will be restored as initial values when the device is turned on. Initial Values in the RAM area are restored when the device is turned on.

Address	Size (Byte)	Data Name	Access	Default Value	Range	Unit
0	2	Model Number	R	1,030	-	-
2	4	Model Information	R	-	-	-
6	1	Firmware Version	R	-	-	-
7	1	ID	RW	1	0 ~ 253	-
8	1	Baud Rate	RW	1	0 ~ 7	-
9	1	Return Delay Time	RW	250	0 ~ 254	2 [μsec]
10	1	Drive Mode	RW	0	0 ~ 1	-
11	1	Operating Mode	RW	3	0 ~ 16	-
12	1	Secondary(Shadow) ID	RW	255	0 ~ 252	-
13	1	Protocol Type	RW	2	1 ~ 2	-
20	4	Homing Offset	RW	0	-1,044,479 ~ 1,044,479	1 [pulse]
24	4	Moving Threshold	RW	10	0 ~ 1,023	0.229 [rev/min]
31	1	Temperature Limit	RW	80	0 ~ 100	1 [°C]
32	2	Max Voltage Limit	RW	160	95 ~ 160	0.1 [V]
34	2	Min Voltage Limit	RW	95	95 ~ 160	0.1 [V]
36	2	PWM Limit	RW	885	0 ~ 885	0.113 [%]
38	2	Current Limit	RW	1,193	0 ~ 1,193	2.69 [mA]
44	4	Velocity Limit	RW	330	0 ~ 1,023	0.229 [rev/min]

For the purpose of torque control, the operating mode of the dynamixel motor needs to be set to **MODE 0**. This can be done using the motor setup code in the git repository. To change the operating mode to **MODE 0**, flash the turtlebot3_setup_motor firmware to the OpenCR board while the motors are connected to the board. Open up a serial terminal with Arduino or Terraterm, under the serial termianl, you will see the following options:

```

1. setup left  motor
2. setup right motor
3. test left   motor
4. test right  motor
5. setup left  motor for current control
6. setup right motor for current control
7. test left   motor for current control
8. test right  motor for current control
>>

```

The first 4 options comes with the OpenCR Arduino library. The last 4 options are added based on the first 4 options and they configure the left and right motors for torque control. For now, the current is limited to half of the highest value to avoid damage to the hardware. If changes are needed, the source code needs to be modified and the motors has to be setup again. To setup the left motor, simply put 5 in the termianl and press enter. Wait until it says it's finished. Then use option 6 for the right motor. After the setup is done, use option 7 and 8 to test the motor. The test option apply a small torque to the motor for one second and stops for one second. Make sure both motors are tested then, flash the turtlebot3_core firmware back. Once the operating mode is set. The OpenCR code is changed to write torque values to the Dynamixel RAM addresses, when the robot is live. The addresses that are useful for this purpose are given in the image below.

102	2	Goal Current	RW	-	-Current Limit(38) ~ Current Limit(38)	2.69 [mA]
104	4	Goal Velocity	RW	-	-Velocity Limit(44) ~ Velocity Limit(44)	0.229 [rev/min]
108	4	Profile Acceleration	RW	0	0 ~ 32,767 0 ~ 32,737	214.577 [rev/min ²] 1 [ms]
112	4	Profile Velocity	RW	0	0 ~ 32,767	0.229 [rev/min]
116	4	Goal Position	RW	-	Min Position Limit(52) ~ Max Position Limit(48)	1 [pulse]
120	2	Realtime Tick	R	-	0 ~ 32,767	1 [msec]
122	1	Moving	R	0	0 ~ 1	-
123	1	Moving Status	R	0	-	-
124	2	Present PWM	R	-	-	-
126	2	Present Current	R	-	-	2.69 [mA]
128	4	Present Velocity	R	-	-	0.229 [rev/min]
132	4	Present Position	R	-	-	1 [pulse]

The code that change the operating mode of the motor is below.

```

bool setupMotorLeftCurrentControl(void)
{
    CMD_SERIAL.println("Setup Motor Left for current control...");

    if (tb3_id < 0)
    {
        CMD_SERIAL.println(" no dxl motors");
    }
    else
    {
        write(portHandler, packetHandler2, tb3_id, 64, 1, 0);
        write(portHandler, packetHandler2, tb3_id, 7, 1, 1);
        tb3_id = 1;
        write(portHandler, packetHandler2, tb3_id, 8, 1, 3);
        portHandler->setBaudRate(1000000);
        write(portHandler, packetHandler2, tb3_id, 10, 1, 0);
        write(portHandler, packetHandler2, tb3_id, 11, 1, 0);

        // this set the limit to half
        write(portHandler, packetHandler2, tb3_id, 38, 2, 596);
        CMD_SERIAL.println(" Warning: Current limit is set to half to protect the motor!");
        CMD_SERIAL.println(" ok");
    }
}

```

```

bool setupMotorRightCurrentControl(void)
{
    CMD_SERIAL.println("Setup Motor Right for current control...");

    if (tb3_id < 0)
    {
        CMD_SERIAL.println(" no dxl motors");
    }
    else
    {
        write(portHandler, packetHandler2, tb3_id, 64, 1, 0);
        write(portHandler, packetHandler2, tb3_id, 7, 1, 2);
        tb3_id = 2;
        write(portHandler, packetHandler2, tb3_id, 8, 1, 3);
        portHandler->setBaudRate(1000000);
        write(portHandler, packetHandler2, tb3_id, 10, 1, 1);
        write(portHandler, packetHandler2, tb3_id, 11, 1, 0);

        // this set the limit to half
        write(portHandler, packetHandler2, tb3_id, 38, 2, 596);
        CMD_SERIAL.println(" Warning: Current limit is set to half to protect the motor!");
        CMD_SERIAL.println(" ok");
    }
}

```

The code used to test the torque control after setup is below.

```

void testMotorCurrentControl(uint8_t id)
{
    uint32_t pre_time;
    uint8_t toggle = 0;

    if (id == 1)
    {
        CMD_SERIAL.printf("Test Motor Left...");
    }
    else
    {
        CMD_SERIAL.printf("Test Motor Right...");
    }
    // We run at 1000000
    portHandler->setBaudRate(1000000);

    uint16_t model_number;
    int dxl_comm_result = packetHandler2->ping(portHandler, id, &model_number);
    if (dxl_comm_result == COMM_SUCCESS)
    {
        CMD_SERIAL.printf(" found type: %d\n", model_number);
        write(portHandler, packetHandler2, id, 64, 1, 1);

        toggle = 0;
        pre_time = millis();
        write(portHandler, packetHandler2, id, 102, 2, 100);
        while (1)
        {
            if (CMD_SERIAL.available())
            {
                flushCmd();
                break;
            }
        }
    }
}

```

```

    if (millis() - pre_time > 1000)
    {
        pre_time = millis();

        toggle ^= 1;

        if (toggle)
        {
            write(portHandler, packetHandler2, id, 102, 2, 0);
        }
        else
        {
            write(portHandler, packetHandler2, id, 102, 2, 100);
        }
    }
    write(portHandler, packetHandler2, id, 102, 2, 0);
}
else
{
    CMD_SERIAL.printf("  dxl motor ID:%d not found\n", id);
}
}

```

The OpenCR Arduino library comes with a TurtleBot3MotorDriver class that helps controlling the 2 motors by commanding velocities. A new class TurtleBot3MotorTorqueDriver was derived (strictly speaking, not inherited, instead just copied and pasted) from the TurtleBot3MotorDriver with new member functions added for controlling the torque instead of velocities. The class is defined as below

```

class TurtleBot3MotorTorqueDriver
{
public:
    TurtleBot3MotorTorqueDriver();
    ~TurtleBot3MotorTorqueDriver();
    bool init(String turtlebot3);
    void close(void);
    bool setTorque(bool onoff);
    bool getTorque();
    bool readEncoder(int32_t &left_value, int32_t &right_value);
    bool writeVelocity(int64_t left_value, int64_t right_value);

    bool writeTorque(int16_t left_value, int16_t right_value);
    bool readTorque(float &left_torque, float &right_torque);
    bool controlMotor(const float wheel_radius, const float wheel_separation, float *value);
    bool controlMotor(float *torque);

private:
    uint32_t baudrate_;
    float protocol_version_;
    uint8_t left_wheel_id_;
    uint8_t right_wheel_id_;
    bool torque_;

    uint16_t dynamixel_limit_max_velocity_;
    uint16_t dynamixel_limit_max_current_;

    dynamixel::PortHandler *portHandler_;
    dynamixel::PacketHandler *packetHandler_;
}

```

```

dynamixel::GroupSyncWrite *groupSyncWriteVelocity_;
dynamixel::GroupSyncRead *groupSyncReadEncoder_;

dynamixel::GroupSyncWrite *groupSyncWriteTorque_;
dynamixel::GroupSyncRead *groupSyncReadTorque_;

```

The following member functions is added to the TurtleBot3MotorDriver class, which takes in input torque value and converts it into appropriate register values for writing to the appropriate addresses.

```

bool TurtleBot3MotorTorqueDriver::controlMotor(float *torque)
{
    bool dxl_comm_result = false;

    int16_t wheel_current_cmd[2];
    wheel_current_cmd[LEFT] = CURRENT_TO_OUTPUT(TORQUE_TO_CURRENT(torque[LEFT]));
    wheel_current_cmd[RIGHT] = CURRENT_TO_OUTPUT(TORQUE_TO_CURRENT(torque[RIGHT]));

    wheel_current_cmd[LEFT] = constrain(wheel_current_cmd[LEFT], -dynamixel_limit_max_current_,
                                         dynamixel_limit_max_current_);
    wheel_current_cmd[RIGHT] = constrain(wheel_current_cmd[RIGHT], -dynamixel_limit_max_current_,
                                         dynamixel_limit_max_current_);

    dxl_comm_result = writeTorque((int16_t)wheel_current_cmd[LEFT], (int16_t)wheel_current_cmd[RIGHT]);

    if (dxl_comm_result == false)
        return false;

    return true;

```

CURRENT_TO_OUTPUT and **TORQUE_TO_CURRENT** are macro functions that does the conversions from torque to current and current to a 2 byte value that the microcontroller understand. All the necessary macros are defined as

```

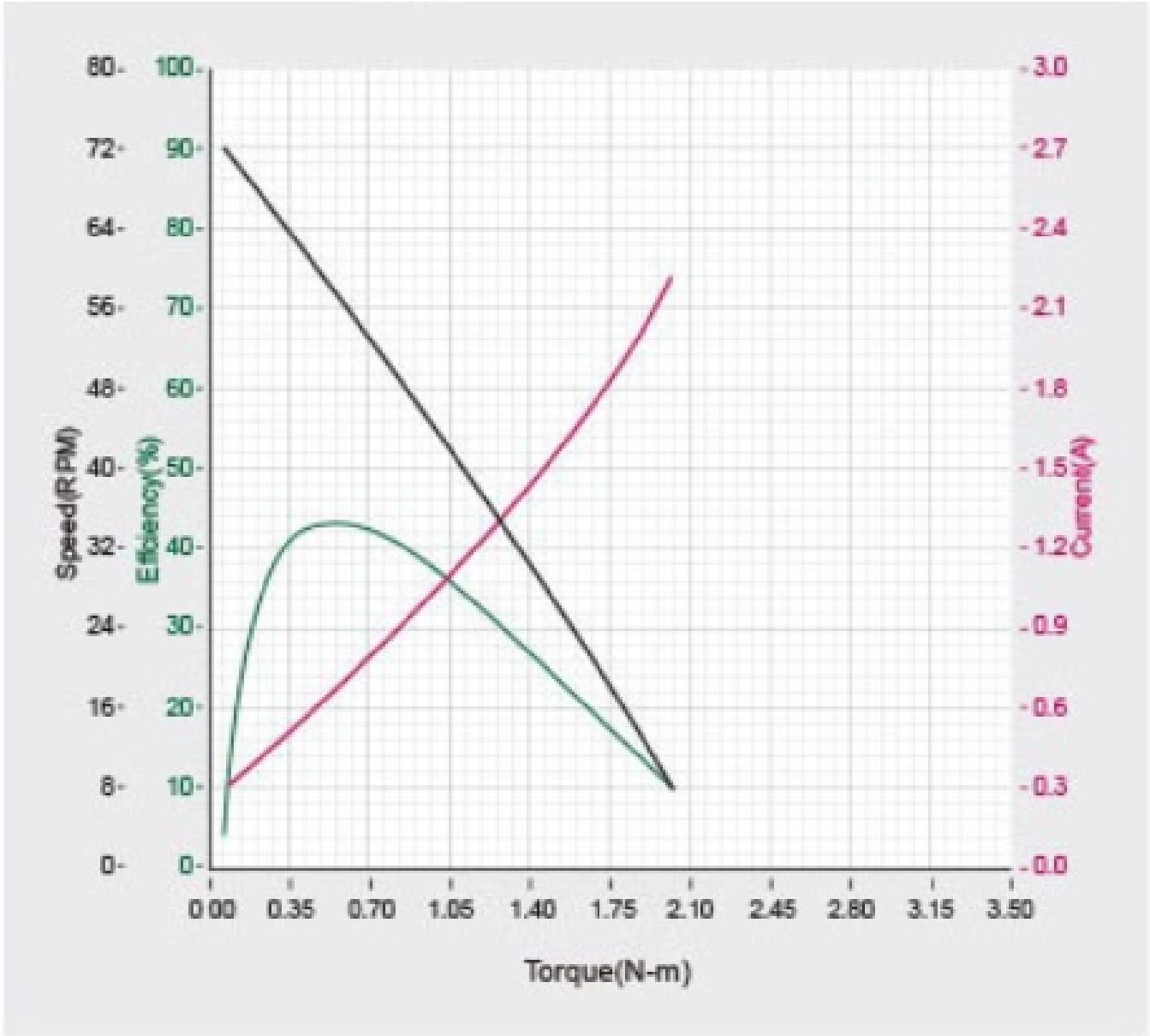
#include <stdint.h>
#define WAFFLE_DXL_LIMIT_MAX_CURRENT 780
#define ADDR_X_GOAL_CURRENT          102
#define ADDR_X_PRESENT_CURRENT       126
#define LEN_X_GOAL_CURRENT           2
#define LEN_X_PRESENT_CURRENT        2

#define MAX_CURRENT_11V1              2.1
#define MAX_TORQUE_11V1               2.7

#define CURRENT_GOAL_UNIT            2.69
#define TORQUE_TO_CURRENT(t)          t * (MAX_CURRENT_11V1 / MAX_TORQUE_11V1) // convert torque to current
                                         in amp
#define CURRENT_TO_OUTPUT(a)          (int16_t)(a * 1000 / CURRENT_GOAL_UNIT)
#define CURRENT_TO_TORQUE(t)          t / (MAX_CURRENT_11V1 / MAX_TORQUE_11V1) // convert current in amp to
                                         torque in N-m

```

This conversion is based on a linear approximation of the actuator performance graph.



This function below takes the converted value and write to both motors using the Dynamixel SDK APIs.

```
bool TurtleBot3MotorTorqueDriver::writeTorque(int16_t left_value, int16_t right_value)
{
    bool dxl_addparam_result;
    int8_t dxl_comm_result;

    dxl_addparam_result = groupSyncWriteTorque_->addParam(left_wheel_id_, (uint8_t *)&left_value);
    if (dxl_addparam_result != true)
        return false;

    dxl_addparam_result = groupSyncWriteTorque_->addParam(right_wheel_id_, (uint8_t *)&right_value);
    if (dxl_addparam_result != true)
        return false;

    dxl_comm_result = groupSyncWriteTorque_->txPacket();
    if (dxl_comm_result != COMM_SUCCESS)
    {
        Serial.println(packetHandler_->getTxRxResult(dxl_comm_result));
        return false;
    }
}
```

```

}

groupSyncWriteTorque_->clearParam();
return true;
}

```

Subscribers are added to subscribe to the topics left_torque right_torque. The following use the function defined above and write the corresponding values to the correct memory location of the Dynamixel motor controller when new message comes in. When there is no new message, it will timeout and write zeros to both motors.

```

if ((t-tTime[0]) >= (1000 / CONTROL_MOTOR_TORQUE_FREQUENCY))
{
    updateGoalTorque();
    //this timeout will stop the motor if no message comes in

    if ((t-tTime[6]) > CONTROL_MOTOR_TIMEOUT)
    {
        motor_driver.controlMotor(zero_torque);
    }
    else {
        motor_driver.controlMotor(torque);
    }
}

```

This Python node will convert the cmd_vel to the left and right velocities command based on the robot geometry.

```

#!/usr/bin/env python2
import rospy

from std_msgs.msg import Float64
from sensor_msgs.msg import Imu, JointState
from math import sin, cos, atan2, sqrt, fabs
from geometry_msgs.msg import Twist

class Driver:
    def __init__(self):
        rospy.init_node('differential_driver')

        self._rate = rospy.get_param('~rate', 500)
        self._wheel_base = rospy.get_param('~wheel_base', 0.28)

        #self._left_speed_percent = 0
        #self._right_speed_percent = 0
        self._left_speed = 0
        self._right_speed = 0

        rospy.Subscriber('cmd_vel', Twist, self.velocity_received_callback)
        self.left_wheel_vel = rospy.Publisher('/left_vel', Float64, queue_size=100)
        self.right_wheel_vel = rospy.Publisher('/right_vel', Float64, queue_size=100)

    def velocity_received_callback(self, message):
        linear = message.linear.x
        angular = message.angular.z

        self.left_speed = linear - angular * self._wheel_base/2
        self.right_speed = linear + angular * self._wheel_base/2

        # print("left wheel velocity:",self.left_speed,"right wheel velocity:",self.right_speed)
        self.publish_velocity()

    def publish_velocity(self):
        self.left_wheel_vel.publish(self.left_speed)

```

```

    self.right_wheel_vel.publish(self.right_speed)

def run(self):
    rate = rospy.Rate(self._rate)
    while not rospy.is_shutdown():
        rate.sleep()

if __name__ == '__main__':
    driver = Driver()
    driver.run()

```

The original Odometry publisher is in the microcontroller. Since the built-in differential drive plugin which provides Odometry that's used in the simulation was replaced by our own controller, An Odometry publisher was added to the system to provide the Odometry in the simulation only. The source code is below.

```

#include <ros/ros.h>
#include <ros/console.h>
#include <sensor_msgs/JointState.h>
#include <sensor_msgs/Imu.h>
#include <math.h>
#include <std_msgs/Float64.h>
#include <tf/transform_broadcaster.h>
#include <geometry_msgs/Quaternion.h>
#include <nav_msgs/Odometry.h>
#include <vector>
#include <string>
#include <iostream>
#include <chrono>
#include <thread>

#define WHEEL_RADIUS           0.033

using std::cout;
using std::endl;

sensor_msgs::Imu imu;
sensor_msgs::JointState lastJointState;
sensor_msgs::JointState currJointState = lastJointState;
geometry_msgs::Quaternion q;
nav_msgs::Odometry odom;
ros::Publisher odom_pub;
geometry_msgs::TransformStamped odom_tf;
std::string odom_header_frame_id;
std::string odom_child_frame_id;
double orientation[4];
double odom_pose[3];
double odom_vel[3];

void init(ros::NodeHandle n);
bool calcOdometry();
void updateOdometry();

bool calcOdometry()
{
    double wheel_l, wheel_r; // rotation value of void updateOdometry(void)wheel [rad]
    double delta_s, theta, delta_theta;
    static double last_theta = 0.0;
    static double lastTime = ros::Time::now().toSec();

```

```

double currTime;
double v, w; // v = translational velocity [m/s], w = rotational velocity [rad/s]
double step_time;

wheel_l = wheel_r = 0.0;
delta_s = delta_theta = theta = 0.0;
v = w = 0.0;
step_time = 0.0;

currTime = ros::Time::now().toSec();
double diff_time = currTime - lastTime;
step_time = diff_time;

if (step_time == 0)
    return false;

wheel_l = (currJointState.position[0] - lastJointState.position[0]);
wheel_r = (currJointState.position[1] - lastJointState.position[1]);

if (isnan(wheel_l))
    wheel_l = 0.0;

if (isnan(wheel_r))
    wheel_r = 0.0;

delta_s = WHEEL_RADIUS * (wheel_r + wheel_l) / 2.0;
q = imu.orientation;
theta = tf::getYaw(q);
delta_theta = theta - last_theta;

// compute odometric pose
odom_pose[0] += delta_s * cos(odom_pose[2] + (delta_theta / 2.0));
odom_pose[1] += delta_s * sin(odom_pose[2] + (delta_theta / 2.0));
odom_pose[2] += delta_theta;
v = delta_s / step_time;
w = delta_theta / step_time;

odom_vel[0] = v;
odom_vel[1] = 0.0;
odom_vel[2] = w;

last_theta = theta;
lastTime = currTime;
return true;
}

void updateOdometry()
{
    odom.header.frame_id = odom_header_frame_id;
    odom.child_frame_id = odom_child_frame_id;

    odom.pose.pose.position.x = odom_pose[0];
    odom.pose.pose.position.y = odom_pose[1];
    odom.pose.pose.position.z = 0;
    odom.pose.pose.orientation = tf::createQuaternionMsgFromYaw(odom_pose[2]);

    odom.twist.twist.linear.x = odom_vel[0];
    odom.twist.twist.angular.z = odom_vel[2];
}

```

```

void updateTF()
{
    odom_tf.header = odom.header;
    odom_tf.child_frame_id = odom.child_frame_id;
    odom_tf.transform.translation.x = odom.pose.pose.position.x;
    odom_tf.transform.translation.y = odom.pose.pose.position.y;
    odom_tf.transform.translation.z = odom.pose.pose.position.z;
    odom_tf.transform.rotation = odom.pose.pose.orientation;
}

void jointStateCallback(const sensor_msgs::JointState::ConstPtr& msg)
{
    static tf::TransformBroadcaster br;
    currJointState = *msg;
    calcOdometry();
    updateOdometry();
    updateTF();
    odom.header.stamp = ros::Time::now();
    odom_pub.publish(odom);
    odom_tf.header.stamp = ros::Time::now();
    br.sendTransform(odom_tf);
    lastJointState = currJointState;

}

void imuCallback(const sensor_msgs::Imu::ConstPtr& msg)
{
    imu = *msg;
}

void init(ros::NodeHandle n)
{
    odom.pose.pose.position.x = 0.0;
    odom.pose.pose.position.y = 0.0;
    odom.pose.pose.position.z = 0.0;

    odom.pose.pose.orientation.x = 0.0;
    odom.pose.pose.orientation.y = 0.0;
    odom.pose.pose.orientation.z = 0.0;
    odom.pose.pose.orientation.w = 0.0;

    odom.twist.twist.linear.x = 0.0;
    odom.twist.twist.angular.z = 0.0;
    std::string get_tf_prefix;
    n.getParam("tf_prefix", get_tf_prefix);
    if (get_tf_prefix == "")
    {
        odom_header_frame_id = "odom";
        odom_child_frame_id = "base_footprint";
    }
    else
    {
        odom_header_frame_id = get_tf_prefix + "/odom";
        odom_child_frame_id = get_tf_prefix + "/base_footprint";
    }
}

int main(int argc, char** argv){
    ros::init(argc, argv, "sim_odom_pub");
    ros::NodeHandle n;
    init(n);
    ros::Subscriber sub = n.subscribe("joint_states", 100 ,jointStateCallback);
}

```

```

ros::Subscriber subImu = n.subscribe("imu", 100, imuCallback);
odom_pub = n.advertise<nav_msgs::Odometry>("odom", 50);
// cout << "init fin" << endl;
ros::spin();
}

```

This program is adapted from the Odometry publisher on the microcontroller. It takes the information from the simulated IMU and joint states to generate Odometry messages for simulation. The orientation is taken from the imu and the position is taken from the joint states. It also publishes the transformations between the base frame and the Odometry frame. The covariance matrix in the Odometry message is ignored and its elements are all 0s.

To control the left and right wheel joint using effort command in the simulation, the URDF and Gazebo descriptions of the robot are also modified as follow.

`src/turtlebot3/turtlebot3_description/urdf/turtlebot3_waffle_pi_effort_controller.urdf.xacro:`

```

<transmission name="tran1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="wheel_left_joint">
    <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor1">
    <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>

<transmission name="tran2">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="wheel_right_joint">
    <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
  </joint>
  <actuator name="motor2">
    <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>

```

`src/turtlebot3/turtlebot3_description/urdf/turtlebot3_waffle_pi_effort_controller.gazebo.xacro:`

```

<gazebo reference="wheel_left_link">
  <mu1>0.75</mu1>
  <mu2>0.75</mu2>
  <kp>500000.0</kp>
  <kd>10.0</kd>
  <minDepth>0.001</minDepth>
  <maxVel>0.1</maxVel>
  <fdir1>1 0 0</fdir1>
  <material>Gazebo/FlatBlack</material>
</gazebo>

<gazebo reference="wheel_right_link">
  <mu1>0.75</mu1>
  <mu2>0.75</mu2>
  <kp>500000.0</kp>
  <kd>10.0</kd>

```

```

<minDepth>0.001</minDepth>
<maxVel>0.1</maxVel>
<fdir1>1 0 0</fdir1>
<material>Gazebo/FlatBlack</material>
</gazebo>

<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
  </plugin>
</gazebo>

<!-- <gazebo>
  <plugin name="turtlebot3_waffle_pi_controller" filename="libgazebo_ros_diff_drive.so">
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <odometryFrame>odom</odometryFrame>
    <odometrySource>world</odometrySource>
    <publishOdomTF>true</publishOdomTF>
    <robotBaseFrame>base_footprint</robotBaseFrame>
    <publishWheelTF>false</publishWheelTF>
    <publishTf>true</publishTf>
    <publishWheelJointState>false</publishWheelJointState>
    <legacyMode>false</legacyMode>
    <updateRate>30</updateRate>
    <leftJoint>wheel_left_joint</leftJoint>
    <rightJoint>wheel_right_joint</rightJoint>
    <wheelSeparation>0.287</wheelSeparation>
    <wheelDiameter>0.066</wheelDiameter>
    <wheelAcceleration>1</wheelAcceleration>
    <wheelTorque>10</wheelTorque>
    <rosDebugLevel>na</rosDebugLevel>
  </plugin>
</gazebo> -->

```

A configuration file was added to make commanding effort to left and right wheel joints in Gazebo simulation possible

/src/ros_control/params/bot_control.yaml

8 Turtlebot Graphical User Interface

This section documents the graphical user interface for manipulating the TurtleBot3 in both real world and simulated environments. It contains descriptions for each graphical widget and code documentation. The `turtlegui` package contains all of the c++ scripts and dependency files responsible for the interface. And once the package is appropriately sourced via the terminal, it can be run with the following command:

`roslaunch turtlebot_gazebo turtlebot_gazebo.launch`

8.1 Qt

The interface runs on Qt, a cross-platform graphical toolkit. This module implements a variety of important classes, the first of which is `QApplication`. The source code for the instantiation of the Turtlebot qt application, in `main.cpp`, is

```
int main(int argc, char **argv) {  
    //*****  
    // Qt  
    //*****  
    QApplication app(argc, argv);  
    turtlegui::MainWindow w(argc, argv);  
    w.show();  
    app.connect(&app, SIGNAL(lastWindowClosed()), &app, SLOT(quit()));  
    int result = app.exec();  
  
    return result;  
}
```

The QApplication class takes care of a lot of things, most notably the input CLI arguments and the event loop. The event loop is a loop that waits for user input in GUI applications, and is launched on this command

```
int result = app.exec();
```

Another important Qt class is QWidget, which is used to describe the widgets in the interface. Widgets are the basic building blocks for a GUI application, and are able to respond to user events. All of the graphical elements in the interface inherit from the QWidget class, including QPushButton, QRadioButton, and QLineEdit. Widgets are also responsible for the handling of the signal and slot mechanism. On a high level, signals and slots are used for communication between objects. In GUI programming, when one widget changes, other widgets need to be notified of this change. Rather than hard-wiring the code that generates these events, signals and slots allow for fluid communication between objects.

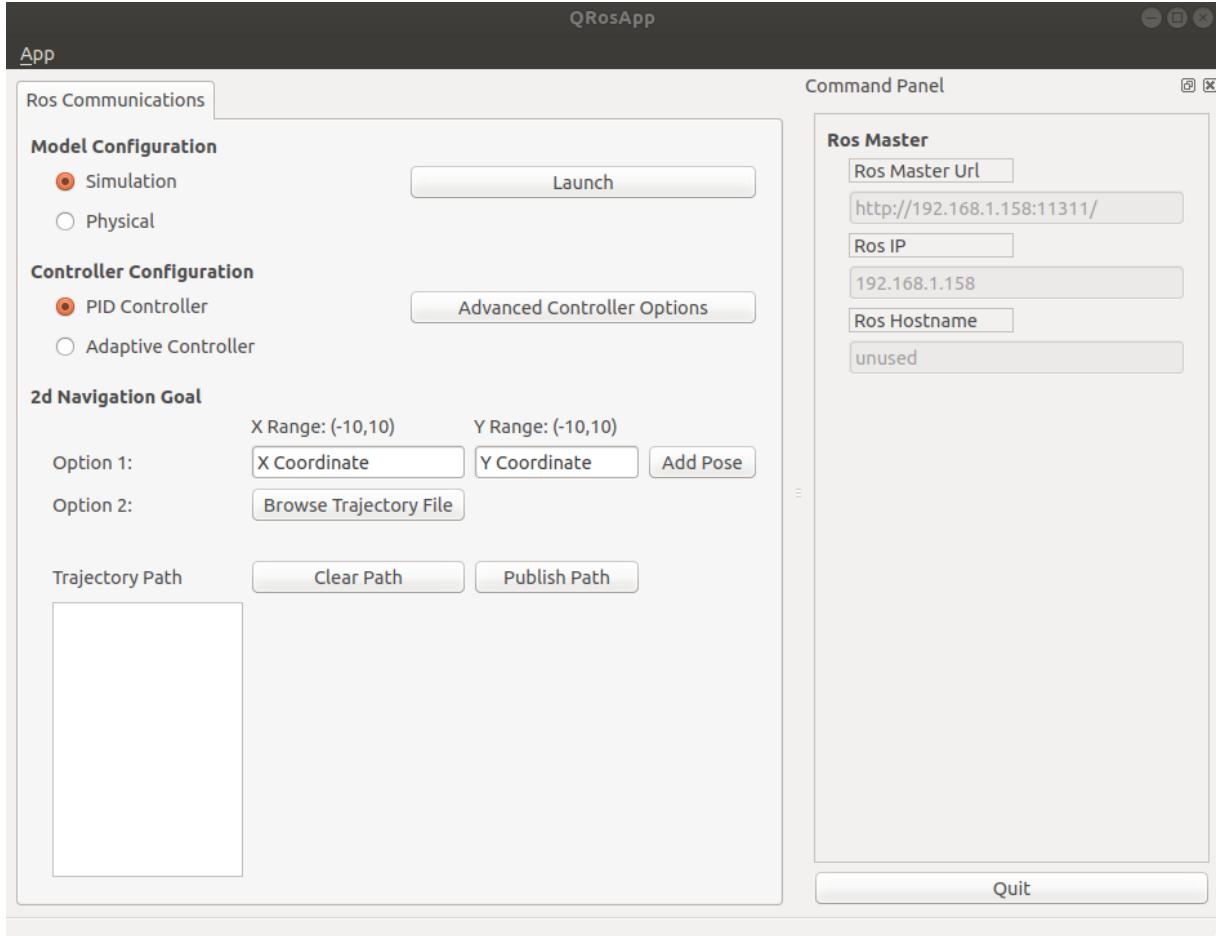
As seen in the line of code below, the lastWindowClosed() signal connects to the quit() slot so that when the exit button is clicked in the GUI, the program terminates.

```
app.connect(&app, SIGNAL(lastWindowClosed()), &app, SLOT(quit()));
```

The Qt toolkit is a very useful module for handling the front-end side of programming and allows for a seamless transition to back-end code.

8.2 GUI Widget Walkthrough

This section gives a detailed description for each widget in main window of the interface. A screenshot of the interface can be seen below



On startup, a ROS Master is initialized, this provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer. All ROS commands are handled implicitly, thus requiring a single terminal window.

The three main sections of the GUI are Model Configuration, Controller Configuration, and 2d Navigation Goal.

8.2.1 Model Configuration

The first tab contains two QRadioButtons and one QPushButton that allow the user to specify and launch an environment for the Turtlebot. The simulation launches Gazebo and RVIZ instances for visualization. This is accomplished by running this ROS command:

```
{
    QMainWindow::closeEvent(event);
```

The ROS launch command is blocking, which means once it is executed, another terminal window must be opened to run any additional commands. The ampersand at the end of the command tells the process to run in the background, allowing for more commands to be run in the same window. The real-world option uses the same syntax but launches a different package:

```
{
    QMainWindow::closeEvent(event);
```

Once an environment is loaded, never before, the dynamics and path of the turtlebot can be manipulated as seen in the following sections. Since there are multiple ROS processes running in the same terminal, ROS has trouble shutting them all down properly and tends to stall the terminal. In order to expedite the shutdown, the timeout

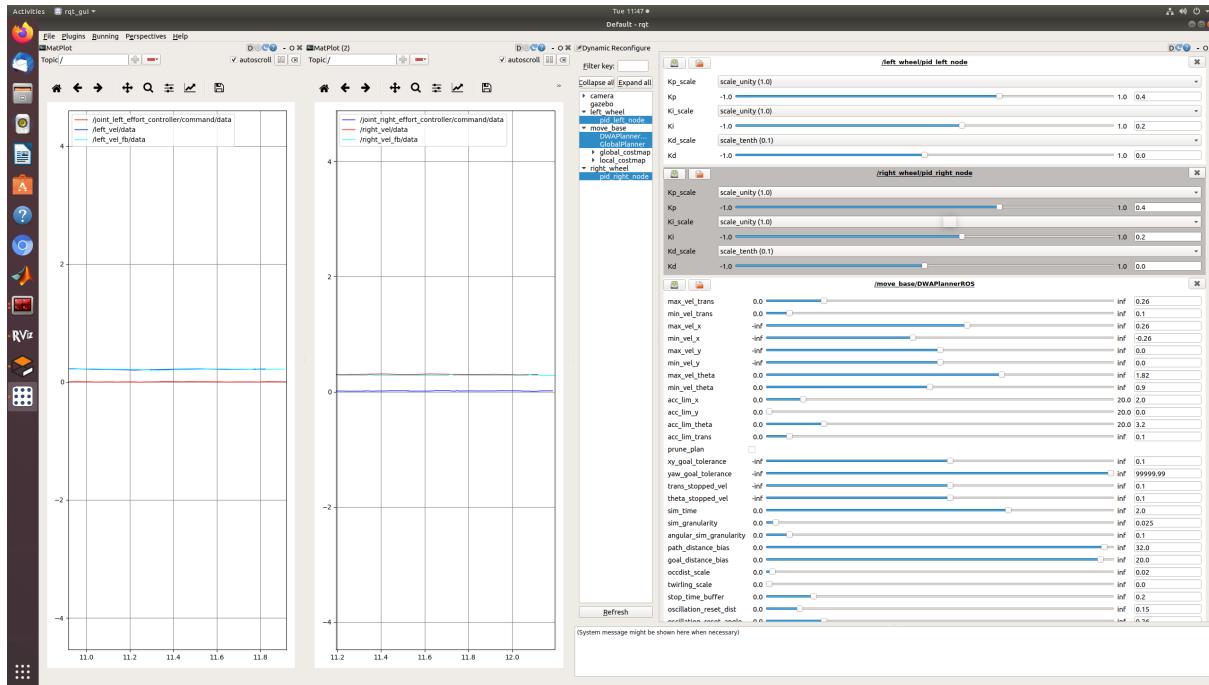
time specified in following file was modified:

```
/opt/ros/melodic/lib/python2.7/dist-packages/roslaunch/nodeprocess.py
```

8.2.2 Controller Configuration

This section allows the user to manipulate the parameters of the turtlebot. Again it contains two QRadiobuttons and one QPushButton for configuring the controller for the Turtlebot. It is important to note that only the PID controller option has been implemented, the Adaptive Controller does not do anything. The update controller parameters button runs on rqt, a software framework of ROS that implements the various GUI tools in the form of plugins. More specifically it uses an rqt metapackage called rqt_gui, which allows multiple rqt plugins to be docked in a single window. The command that handles the instantiation and reference to this framework is:

This launches a new window for viewing all ROS nodes and topics, configuring the PID parameters, and graphically analyzing the error of the PID controller. A screenshot of the window is shown below



8.2.3 2d Navigation Goal

Note that on a new computer, when rqt_gui run in the first time, it will be a blank screen, the user would have to import a perspective manually to load this pre-configured perspective. Simply follow the steps below to load the perspective:

From the turtlegui, select "Simulation" then click "Launch"

Click "Advanced Controller Parameters"

In the rqt_gui windows that poped-up, click "Perspective" on the top menu bar, then click Import to browse for **rqt_gui.perspective** located under the root of the ros workspace.

The turtlebot is able to follow one or more precise waypoints specified by the user via the 2D Navigation Goal section of the interface. Option 1 allows for the publishing of a single waypoint while option 2 lets the user browse for a csv file containing a list of waypoints. The trajectory path inherits the QStringListModel class that keeps track of the robot's current path. Once either option is chosen, the trajectory path will be updated. The clear_path button simply removes all items from the trajectory path. To get the Turtlebot to actually start moving and following the path, the publish_path button must be pressed.

This button utilizes the `follow_waypoints` package, which handles the low-level robot communication for following a path. Three critical ROS commands are executed, the first of which can be seen below.

```
while (ss)
{
```

This creates three ROS topics, /initialpose, /path_ready, and /path_reset. The /path_reset topic is published when the clear path button is clicked. The input trajectory path is published to the /initialpose topic. This topic is of type geometry_msgs/PoseWithCovarianceStamped, thus the messages published to this topic have to be of this type. This message type contains two fields, std_msgs/Header header and geometry_msgs/PoseWithCovariance pose. The former field is just used for communicate timestamped data in a particular coordinate frame. The latter has two more implicit fields, geometry_msgs/Pose pose and float64[36] covariance. The pose field represents a pose in free space with uncertainty. The covariance field is a row-major representation of the 6x6 covariance matrix, which specifies the uncertainty of the pose field. The pose field has two more implicit fields, geometry_msgs/Point position and geometry_msgs/Quaternion orientation. The former contains the position of a point in free space, with fields: float64 x, float64 y, float64 z. The latter represents an orientation in free space in quaternion form, with fields float64 x, float64 y, float64 z, float64 w. By default, the covariance, orientation, and pose.z are all zero, thus the user only needs to specify an x and y position. It is important to note that if the specified goal is not close to current position of the Turtlebot, it will not move. The trajectory.csv file can be used to get an idea of the coordinate pairs the Turtlebot will be able to move to.

Once the path is published, another ROS command is executed:

```
y = pose[1];
```

This just publishes an empty message to /path_ready which tells the Turtlebot to start following the path published to /initialpose. Once the Turtlebot reaches the last point in the path, it will stop moving and a new path will have to be specified. The Turtlebot always starts at the origin, and its position is continuously published to the /odom ROS topic.

9 List of ROS Nodes, Topics, Parameters, Transform

9.1 List of Important ROS Nodes

- /differential_driver
- /left_wheel/pid_left_node
- /message_redirect
- /move_base
- /odom_pub
- /right_wheel/pid_right_node
- /robot_state_publisher
- /rviz
- /turtlebot3_slam_gmapping
- /follow_waypoints

9.2 List of Topics and Their Types

- /map_metadata [nav_msgs/MapMetaData]
- /camera/rgb/image_raw [sensor_msgs/Image]
- /move_base/current_goal [geometry_msgs/PoseStamped]
- /move_base/local_costmap/footprint [geometry_msgs/PolygonStamped]
- /tf [tf2_msgs/TFMessage]
- /move_base/local_costmap/costmap [nav_msgs/OccupancyGrid]
- /odom [nav_msgs/Odometry]
- /joint_left_effort_controller/command [std_msgs/Float64]
- /scan [sensor_msgs/LaserScan]
- /left_vel_fb [std_msgs/Float64]
- /camera/rgb/image_raw/compressed [sensor_msgs/CompressedImage]
- /left_wheel/pid_left_node/parameter_updates [dynamic_reconfigure/Config]
- /move_base_simple/goal [geometry_msgs/PoseStamped]
- /move_base/DWAPlannerROS/cost_cloud [sensor_msgs/PointCloud2]
- /move_base/DWAPlannerROS/trajectory_cloud [sensor_msgs/PointCloud2]
- /tf_static [tf2_msgs/TFMessage]
- /imu [sensor_msgs/Imu]
- /map [nav_msgs/OccupancyGrid]
- /move_base/global_costmap/footprint [geometry_msgs/PolygonStamped]
- /joint_right_effort_controller/command [std_msgs/Float64]
- /cmd_vel [geometry_msgs/Twist]

- /move_base/DWAPlannerROS/local_plan [nav_msgs/Path]
- /waypoints [geometry_msgs/PoseArray]
- /joint_states [sensor_msgs/JointState]
- /move_base/goal [move_base_msgs/MoveBaseActionGoal]
- /initialpose [geometry_msgs/PoseWithCovarianceStamped]
- /rosout_agg [rosgraph_msgs/Log]
- /move_base/GlobalPlanner/plan [nav_msgs/Path]
- /move_base/DWAPlannerROS/global_plan [nav_msgs/Path]
- /left_vel [std_msgs/Float64]
- /right_wheel/pid_debug [std_msgs/Float64MultiArray]
- /camera/rgb/image_raw/compressedDepth [sensor_msgs/CompressedImage]
- /camera/rgb/camera_info [sensor_msgs/CameraInfo]
- /camera/rgb/image_raw/theora [theora_image_transport/Packet]
- /gazebo/link_states [gazebo_msgs/LinkStates]
- /move_base/GlobalPlanner/potential [nav_msgs/OccupancyGrid]
- /gazebo/model_states [gazebo_msgs/ModelStates]
- /clock [rosgraph_msgs/Clock]
- /right_vel_fb [std_msgs/Float64]
- /move_base/cancel [actionlib_msgs/GoalID]
- /right_vel [std_msgs/Float64]
- /move_base/global_costmap/costmap [nav_msgs/OccupancyGrid]
- /left_torque [std_msgs/Float64]
- /right_torque [std_msgs/Float64]
- /battery_state [sensor_msgs/BatteryState]

9.3 List of Important Topics and Descriptions

9.4 List of Important Parameters and How to Find Them

Topic Name	Index
/camera/rgb/camera_info	1
/camera/rgb/image_raw	2
/camera/rgb/image_raw/compressed	3
/camera/rgb/image_raw/theora	4
/cmd_vel	5
/imu	6
/initialpose	7
/joint_left_effort_controller/command	8
/joint_right_effort_controller/command	9
/joint_states	10
/left_vel	11
/left_vel_fb	12
/left_wheel/pid_enable	13
/map	14
/map_metadata	15
/move_base/DWAPlannerROS/local_plan	16
/move_base/GlobalPlanner/plan	17
/move_base/global_costmap/costmap	18
/move_base/goal	19
/move_base/local_costmap/costmap	20
/odom	21
/path_ready	22
/path_reset	23
/right_vel	24
/right_vel_fb	25
/right_wheel/pid_enable	26
/scan	27
/waypoints	28
/left_torque	29
/right_torque	30
/battery_state	31

Index	Descriptions
1	http://docs.ros.org/melodic/api/sensor_msgs/html/msg/CameraInfo.html
2	Raw camera data directly from sensor
3	Compressed image
4	Compressed image based on therora encoding
5	Command Velocity from ROS navigation stack
6	IMU Information (acceleration, orientation)
7	Originally initial pose estimate of AMCL, instead used for adding waypoints
8	Left effort command (only in simulation)
9	Right effort command (only in simulation)
10	Left and right wheel joint states (angular velocity, angular position)
11	Left wheel velocity command
12	Left wheel velocity feedback
13	Left wheel PID controller enable
14	Map drawn by SLAM algorithm
15	Map mata information (width, height, resoulution)
16	Local trajectory to follow by robot generated by the local planner (uses odom frame)
17	Global plan generated by the global planner
18	Global costmap generated from SLAM map and lidar (uses map frame)
19	Current goal to reach
20	Local costmap generated from part of the global costmap and lidar (uses odom frame)
21	Odometry of the robot (accumulated orientation and position based on the feedabck of the motor and IMU)
22	Singal to start following waypoints
23	Singla to reset the waypoints
24	Right wheel velocity command
25	Right wheel velocity feedback
26	Right wheel PID controller enable
27	Data from lidar (in terms of angle and distance)
28	Array of waypoints for visualization purpose
29	Left torque command (only in physical robot)
30	Right torque command (only in physical robot)
31	Battery voltage, designed capacity, percentage left (only in physical robot)

Parameters

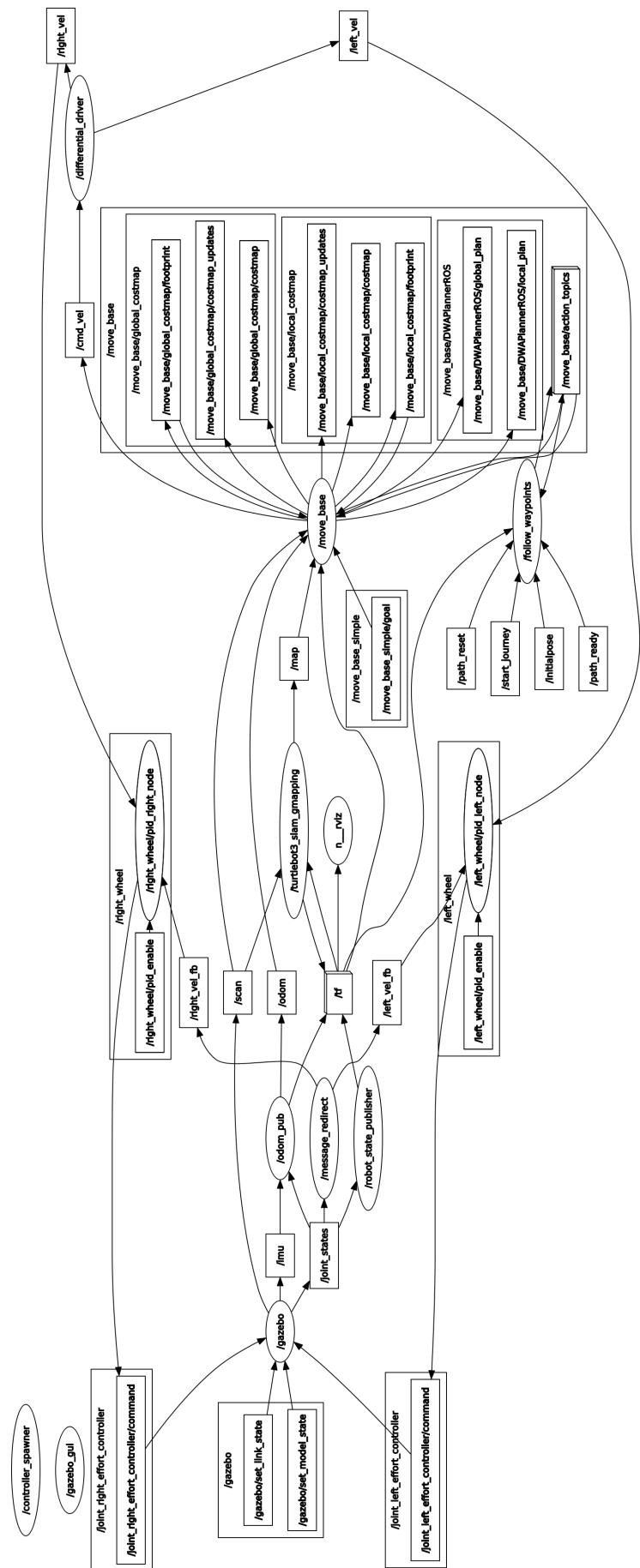
Kp, Ki, Kd in simulation
Kp, Ki, Kd in physical robot
Local planner parameters

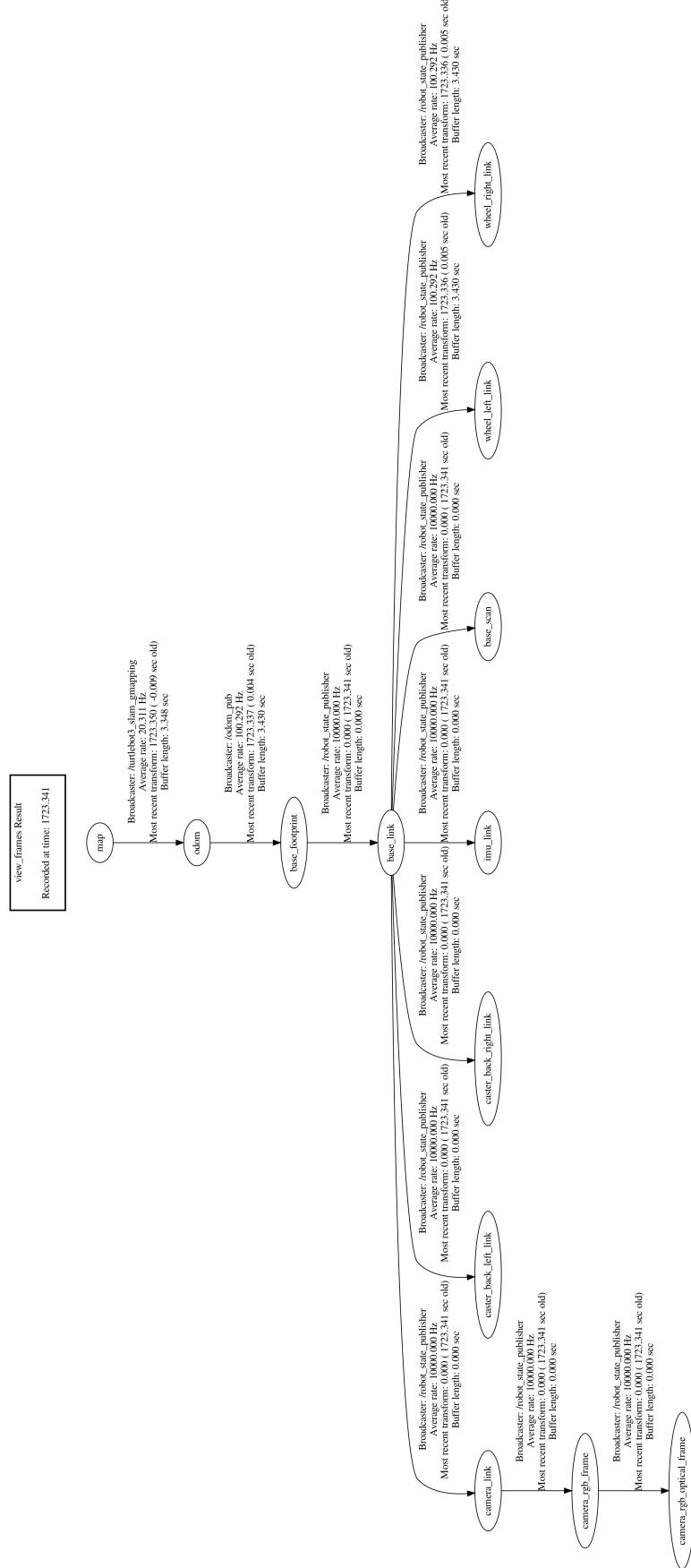
Global planner parameters
move_base parameters
Cartographer parameters physical robot
Cartographer parameters in simulation
Hector SLAM parameters
AMCL parameters
Gmapping parameters
Common costmap parameters

Global costmap parameters
Local costmap parameters
SLAM Method

Location

src/ros_control/launch/turtlebot3_control.launch
src/ros_control/launch/turtlebot3_control_simulation.launch
src/turtlebot3/turtlebot3_navigation/
param/dwa_local_planner_params_waffle_pi_effort_controller.yaml
src/turtlebot3/turtlebot3_navigation/param/global_planner_params.yaml
src/turtlebot3/turtlebot3_navigation/param/move_base_params.yaml
src/turtlebot3/turtlebot3_slam/config/turtlebot3_lds_2d.lua
src/turtlebot3/turtlebot3_slam/config/turtlebot3_lds_2d_gazebo.lua
src/turtlebot3/turtlebot3_slam/launch/turtlebot3_hector.launch
src/turtlebot3/turtlebot3_navigation/launch/amcl.launch
src/turtlebot3/turtlebot3_slam/config/gmapping_params.yaml
src/turtlebot3/turtlebot3_navigation/param/
costmap_common_params_waffle_pi_effort_controller.yaml
src/turtlebot3/turtlebot3_navigation/param/global_costmap_params.yaml
src/turtlebot3/turtlebot3_navigation/param/local_costmap_params.yaml
src/turtlebot3/turtlebot3_navigation/launch/turtlebot3_navigation_no_map.launch





10 Transferring PID Controller to SBC [Experimental Future Work]

The entire path planning and navigation responsibility is taken care of by the Remote PC in the current configuration of the software. The performance of the software is really good, and no network latency issues were identified. However, it is possible to transfer the control software from the Remote PC to the SBC. **This implementation has not been tested to work on the TurtleBot3.** Transferring any software from Remote PC to SBC requires three things:

- Compiling custom packages, if any, associated with that software in the same workspace of the stock packages.
- Installing required dependencies on the SBC, if they are available for the linux distro and ROS version running on the TurtleBot SBC.
- Changing the `*.launch` files on the remote PC and SBC

The control on the TurtleBot3 is achieved through the package `ros_control`. To run the controller on the SBC, copy the package to the `catkin_ws` folder on the SBC. The dependencies required for the package are listed in the `CMakeLists.txt` file.

```
find_package(catkin REQUIRED COMPONENTS
    roscpp
    rospy
    std_msgs
    sensor_msgs
    tf
    nav_msgs
    geometry_msgs
)
```

Finally re-compile the workspace by running `catkin_make` in the base directory (by default, `/catkin_ws/`). In addition to the above dependencies, the SBC should have the `ros-pid` package installed. Run the following command to install it on the SBC.

```
sudo apt-get install ros-kinetic-pid
```

This completes step 1 and 2 of the process. Lastly, we need to modify the appropriate `*.launch` files. Open `turtlebot3_robot.launch` file from the `turtlebot3_bringup` package using the following command.

```
rosed turtlebot3_bringup turtlebot3_robot.launch
```

The file should look something like this.

```
<launch>
  <arg name="multi_robot_name" default="" />
  <arg name="set_lidar_frame_id" default="base_scan" />

  <include file="$(find turtlebot3_bringup)/launch/turtlebot3_core.launch">
    <arg name="multi_robot_name" value="$(arg multi_robot_name)" />
  </include>
  <include file="$(find turtlebot3_bringup)/launch/turtlebot3_lidar.launch">
    <arg name="set_frame_id" value="$(arg set_lidar_frame_id)" />
  </include>

  <node pkg="turtlebot3_bringup" type="turtlebot3_diagnostics" name="turtlebot3_diagnostics"
        output="screen" />
</launch>
```

Copy the following lines from the `turtlebot3_control.launch` file, and append it to the above file before the end tag `</launch>`.

```

<node name="differential_driver" pkg="ros_control" type="differential_driver.py" />

<node name="message_redirect" pkg="ros_control" type="ros_control_node" />

<node name="pid_left_node" pkg="pid" type="controller" ns="left_wheel">
    <param name="Kp" value="0.56" />
    <param name="Ki" value="0.84" />
    <param name="Kd" value="0.0" />
    <param name="upper_limit" value="2.7" />
    <param name="lower_limit" value="-2.7" />
    <param name="windup_limit" value="10" />
    <param name="max_loop_frequency" value="300.0" />
    <param name="min_loop_frequency" value="150.0" />
    <param name="node_name" value="pid_left_node" />
    <param name="topic_from_controller" value="/left_torque" />
    <param name="setpoint_topic" value="/left_vel" />
    <param name="topic_from_plant" value="/left_vel_fb" />

</node>

<node name="pid_right_node" pkg="pid" type="controller" ns="right_wheel">
    <param name="Kp" value="0.56" />
    <param name="Ki" value="0.84" />
    <param name="Kd" value="0.0" />
    <param name="upper_limit" value="2.7" />
    <param name="lower_limit" value="-2.7" />
    <param name="windup_limit" value="10" />
    <param name="max_loop_frequency" value="300.0" />
    <param name="min_loop_frequency" value="150.0" />
    <param name="node_name" value="pid_right_node" />
    <param name="topic_from_controller" value="/right_torque" />
    <param name="setpoint_topic" value="/right_vel" />
    <param name="topic_from_plant" value="/right_vel_fb" />

</node>
```

Finally, edit the launch file `turtlebot3_physical_nav_control.launch` located in the `turtlebot3_bringup` package in the Remote PC and remove the following line from the file.

```
<include file="$(find ros_control)/launch/turtlebot3_control.launch" />
```

This should enable `ros-pid` node and associated custom nodes to run on the SBC.